Introduction to Artificial Intelligence – Final Paper

William Horn
wbhorn@alaska.edu

Hal Dimarchi
hadimarchi@alaska.edu

Duane Shaffer
dlshaffer@alaska.edu

Kai Davids
kdavidsschell@alaska.edu

May 3, 2018

1 Checkers

For the beginning of the project, we created a program by making a move generator and a random move selector. The move generator was used to get valid moves on an 8x8 checkers board at any point in the game. Internally this corresponded to a list of characters of size 32 encoded in the style:

"rrrrrrrrrrrr________bbbbbbbbbbbb"

for a starting board where each underscore represents a space character. Red checkers were represented by 'r', black by 'b' and kings by 'R' and 'B'. Empty spaces were represented by spaces – this led to a small amount of extra work when it came to connecting with skynet, which used '_' to represent empty spaces.The move selector takes moves from the generator, if a jump is present it must be taken. This left us with creating players that could use the board and move generator to play a checkers game.

The game itself has two players, red and black. The players keep track of their pieces and what those pieces can do. The move generators gives valid moves that the player class uses when selecting a move, initially the moves were picked at random. Further improvements on the checkers program were to create a piece count evaluator that chose moves by traversing the search tree of possible moves (see minimax) and picking the move that lead to the best outcome. The heuristic used to evaluate (PieceCount) was to sum up the pieces on the board for each player, giving a larger weight to kings, and taking the difference between the two players. This ended up being quite a good heuristic for evaluating moves and lead to more intelligent move selection.

Fig. 3 and Table 3. show the checkers game running along with the GUI. The table shows all the valid possible moves and jumps at the point of the screenshot. The numbers in the table correspond to the location in Fig. 2.

Table 1: Minimax Statistics

	2	**4**	**6**	**8**
boards expanded/move	7	1051	114748	11237947
evaluations	49	4242	493578	52875269
inner vs leaf nodes	.14	.25	.23	.21

2 Searching

Searching is done to select the move that will have the highest chance of leading to a win for the searching player. This can be done using either a breadth first search (BFS) or a depth first search (DFS). To save on memory we used a DFS.

2.1 Minimax

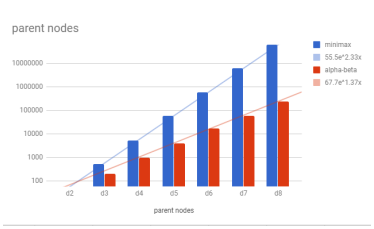
In minimax searching the program starts with all moves currently available to it, and expands them, by assuming they have been taken, and then recursing on the resulting available moves to a chosen depth. The search alternates between choosing the move that will result in the best case scenario for the searching player, and the move that will result in the worst case scenario for the searching player (assuming that the enemy will choose those moves). Once it has reached the max depth or the game is over, it performs an evaluation of the state of the board and returns it. The program chooses the move that leads to the sub-tree with the best possible result. Table 1 shows the statistics for the amount/types of node expanded durning minimax search. This implimentation of minimax only counts boards expanded with moves as adding towards the depth of the search.

2.2 Alpha-Beta

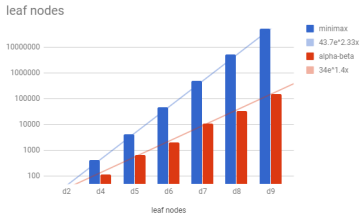
Alpha-beta pruning is an improvement on the minimax searching algorithm that removes branches that are unlikely to be useful. It does so by keeping track of 2 values, alpha, the best explored path to the root for the maximiser, and beta, the best explored path to the root for the minimizer. When expanding nodes and looking at paths for the maximizer, if the program finds an option that will allow the maximizer to achieve a higher value path back to the root than a path already explored then all children of the parent of that leaf node can be pruned, because the minimizer presumably will not choose that option. Because this means whole subtrees don't need to be fully expanded, it allows for either deeper searches or for a more computationally intensive board evaluation function. The graphs in Fig. 1 show the number of nodes expanded for minimax with and without alpha-beta pruning at different depths. Also Table 2 shows the exact number of nodes evaluated at each depth for both minimax and alpha beta search.

Table 2: Statistics on alphabeta vs. minimax search algorithms

	2	**4**	**6**	**8**
minimax	56	5293	607326	64113216
alphabeta	56	960	17471	245591
pruned %	0.00	0.82	0.97	0.99



(a) parent (inner) nodes evaluated in minimax vs alpha-beta



(b) Comparing leaf nodes evaluated in minimax vs alpha-beta

Figure 1: Comparing minimax and alpha-beta search methods.

2.3 Implementation

To implement alpha-beta the same checkers-game class used to play a regular game was used to search through possible future moves and jumps. It was implimented using a recursive DFS search Fig. 1.

Preprocessor macros are used for two reasons (shown in Listing 1), the best way to write the loop through all the action (moves/jumps) is to take the `isMaximizingPlayer` check out of the loop, which means writing it twice and because instead of recursing on boards, move and jumps are used to recures on. This means two very similar function because they are different types.

In the `SEARCH_BASED_ON_MAXIMIZING_PLAYER` (Listing 1, Lines 20–25) macro the actions is either a list of valid moves or jumps depending on what function the macro is expanded in. The `depthExpr` is either n or n - 1. This is done because when recursing on jumps or multi jumps the branch factor is low.

Time permitting this version will be switched to a version that recurses on boards so that iterative deepening search is easier to implement. Currently when a piece makes a multi–jump this search is called multiple times for each jump, instead of once for the whole jump. Recursing on a board would negate this because the entire multi–jump would be counted as one move.

3 Neural Networks

A basic feed forward neural network (NN) consists of an input layer with a variable amount of hidden layers and an output layer. All of the layers consist of nodes and these nodes are connected from layer to layer. The input layer

consists of the data being given to the network that has been formatted to numbers. Each node in the next layer has an amount of weights equal to the number of input nodes. Further nodes have a number of weights equal the size of the layer before them. To calculate a node, multiply each of the previous layer nodes by their associated weights, add them together, and multiply by an activator function. This process is repeated for each layer, causing the values of each node to feed farther forward in the network culminating in a final value for the output. Listing 2 shows the c++ that implements this technique. The outermost for loop (Listing 2, Line 3) is used for selecting different NN at different stages in the game. The stage of the game is determined by the number of pieces on the board. The performance of the NN code is measured in board evaluations per second (BPS).

3.1 Timing

- Blondie24 (32-40-10-1) : 879,833 bps
- Larger (32-1000-100-1): 13762.5 bps

3.2 Evolution (3-10-1 Topology)

Fig. 4 shows the contents of a random NN following the given topology. Fig. 5 shows values in the offspring from the parent in previous image (the evolved network).

In order to show that our offspring is correct, a histogram was made of the change in weight value between parent and offspring. The changes show a model Gaussian distribution. The evolution process was simply following formulas given. Between graphical representation of the change in weights and visual confirmation of viable values for other parameters of the network. This proves that the evolution process was happening properly.

To prove that the Gaussian pseudo-random number generator is absolutely correct, Fig. 7 a histogram of 1000 numbers was made and this confirmed that our generator was in fact Gaussian.

4 Evolutionary Learning

There are several design considerations for evolutionary learning that varies between the members of our group. For the trial section of our project, we used elements from Blondie24's experiment. This includes a NN topology of 32-40-10-1, a population size of 30, and a survival of the fittest method of evolution where the best half of the networks create offspring, evolve, and replace the lower half of the networks. Not all of our members have decided on the topology they want to evolve but one experimental topology is 32-1000-1 with a population size of 100. All networks follow the Blondie24 method of evolution.

Networks are matched up against each other with every network playing 5 games as red side versus a random network on black side. Winners get one

point and losers lose 2. Nothing happens to a NNs performance when a draw occurs. These games compose a points based tournament that allows networks to be sorted based on their performance.

We made several changes to the input of our networks over the course of its evolution. At the start we were not accounting for a piece count value, which resulted in the networks having poor performance vs a pure PieceCount player. However with the addition of the PieceCount value the networks began consistently beating pure PieceCount. One trend we noticed in watching our networks play games vs the networks of other teams in the class was that our networks seemed to value kings equal to regular pieces, and on checking the value we found that evolution had been pushing the value to below 1 (the value of a standard piece). This led us to implement a limit on the value, keeping it within a range from 1-3, as done in the Blondie24 experiment. For the network used in the final tournament our king value was 1.4.

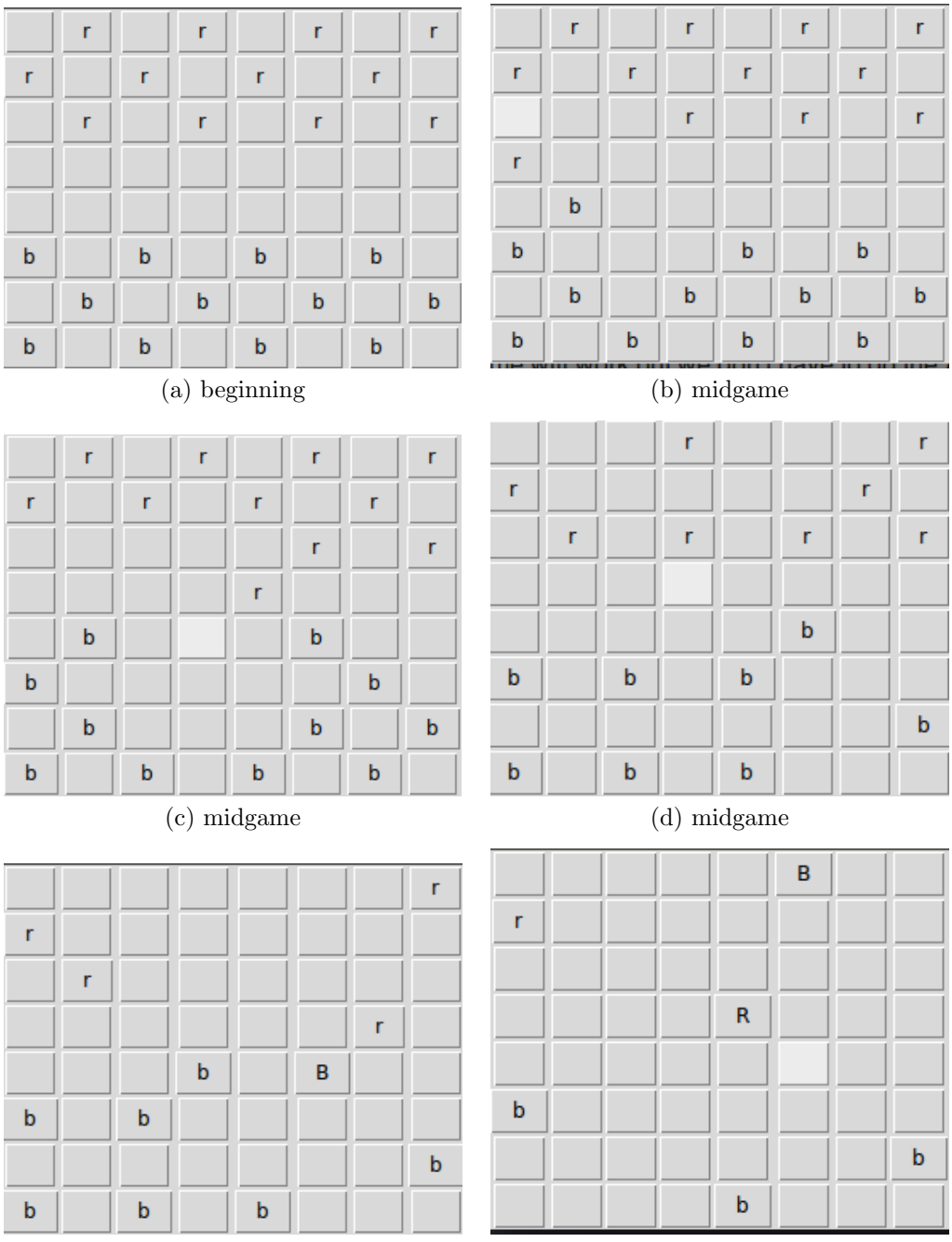
4.1 Tournament Statistics

One of the testing topologies used was a 32-200-20 NN. For this tourament we only played 3 games per match and ran at a alpha-beta search depth of 6. The tournaments were run on a Digital Ocean cloud server running with 2 vCPUs. Threading was done on a per game level so generations could be done faster. The time below is normalized for 1 vCPU and does not taken into account the time benefits from threading at a generation level.

- Time For a Game : 13.53 seconds
- Time For a Generation : 1217.04 seconds
- Generations Per Day : 71.00 gpd

```
/*         0   1   2   3   4   5   6   7
          +---+---+---+---+---+---+---+---+
      0   | X | 0 |   | 1 |   | 2 |   | 3 |
          +---+---+---+---+---+---+---+---+
      1   | 4 |   | 5 |   | 6 |   | 7 |   |
          +---+---+---+---+---+---+---+---+
      2   |   | 8 |   | 9 |   | 10|   | 11|
          +---+---+---+---+---+---+---+---+
      3   | 12|   | 13|   | 14|   | 15|   |
          +---+---+---+---+---+---+---+---+
      4   |   | 16|   | 17|   | 18|   | 19|
          +---+---+---+---+---+---+---+---+
      5   | 20|   | 21|   | 22|   | 23|   |
          +---+---+---+---+---+---+---+---+
      6   |   | 24|   | 25|   | 26|   | 27|
          +---+---+---+---+---+---+---+---+
      7   | 28|   | 29|   | 30|   | 31| X |
          +---+---+---+---+---+---+---+---+
*/
```

Figure 2: Numbering system used to store board states internally.



(a) beginning (b) midgame

(c) midgame (d) midgame

(e) midgame (f) endgame

Figure 3: Screenshots of the GUI at different stages in the game.

Table 3: All possible valid moves and jumps for the gui screenshots in Fig. 3

Move/Jump Generator Output For Black								
Fig 1a.			**Fig 1b.**			**Fig 1c.**		
Moves	**From**	**To**	**Moves**	**From**	**To**	**Jumps**	**From**	**Thru, To**
	20	16		16	13		18	(14, 9)
	21	16, 17		22	17, 18	**Moves**	**From**	**To**
	22	17, 18		23	18, 19		16	12, 13
	23	18, 19		24	21		18	15
				25	21		23	19
							24	21
							26	22
							29	25
							30	25
Fig 1d.			**Fig 1e.**			**Fig 1f.**		
Moves	**From**	**To**	**Jumps**	**From**	**Thru, To**	**Moves**	**From**	**To**
	18	14, 15		18	(15, 11)		2	6, 7
	20	16	**Moves**	**From**	**To**		20	16
	21	16, 17		17	13, 14		27	23
	22	17		18	14, 22, 23		30	25, 26
	27	23		20	16			
	28	24		21	16			
	29	24, 25		27	23			
	30	25, 26		28	24			
				29	24, 25			
				30	25, 26			

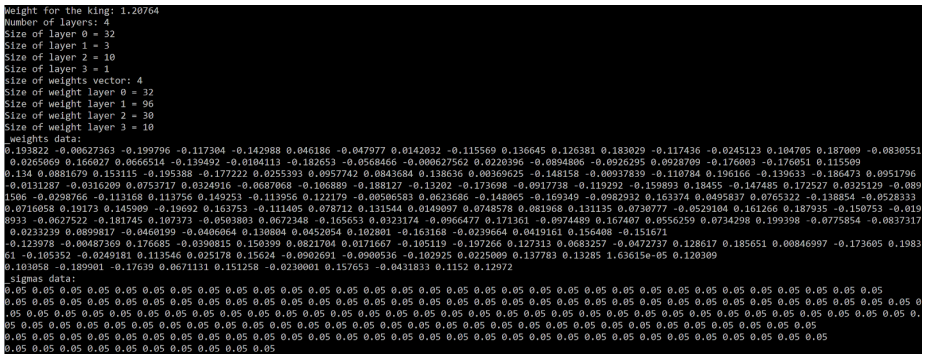


Figure 4: values before

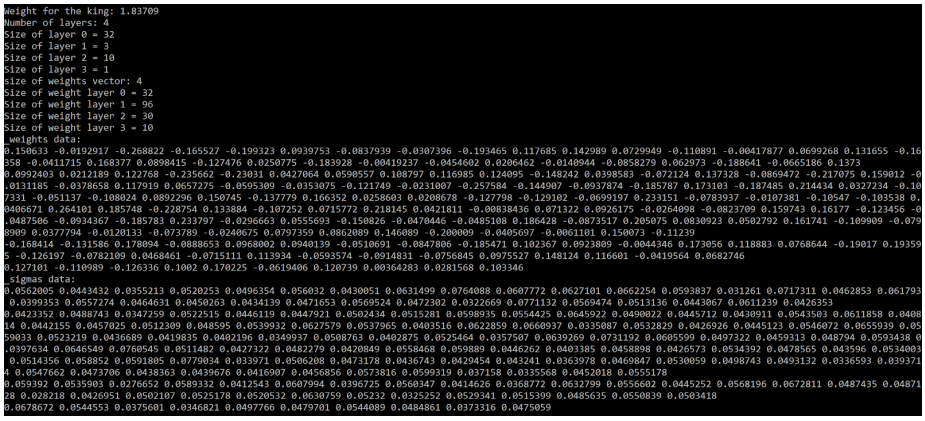


Figure 5: values after

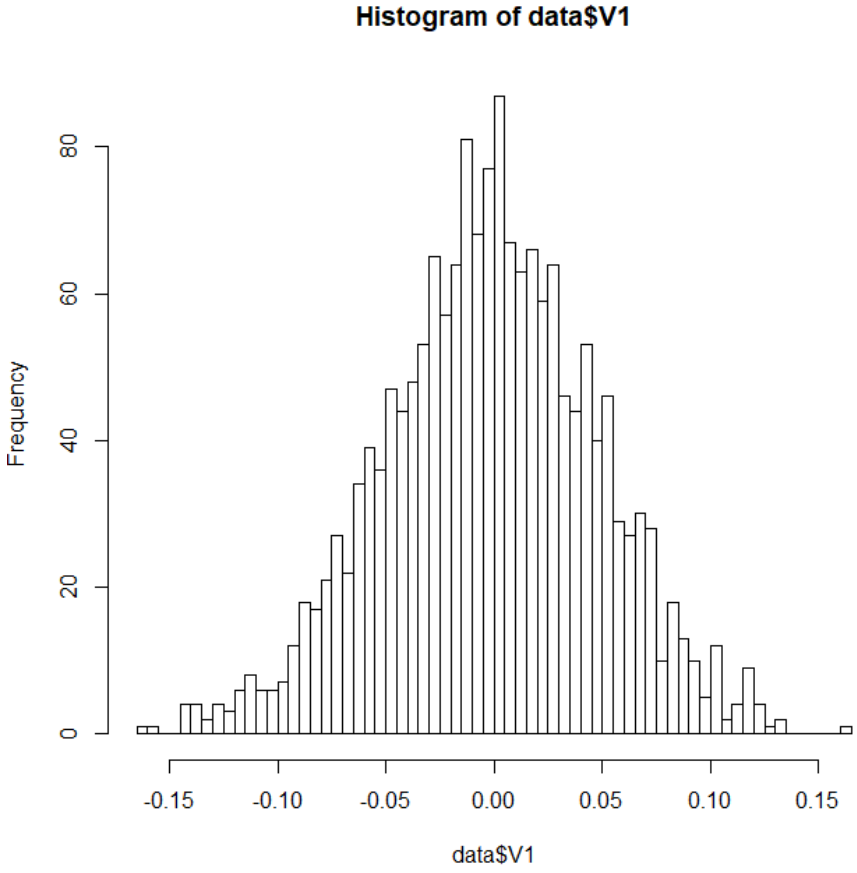


Figure 6: histogram of Gaussian numbers used for evolution

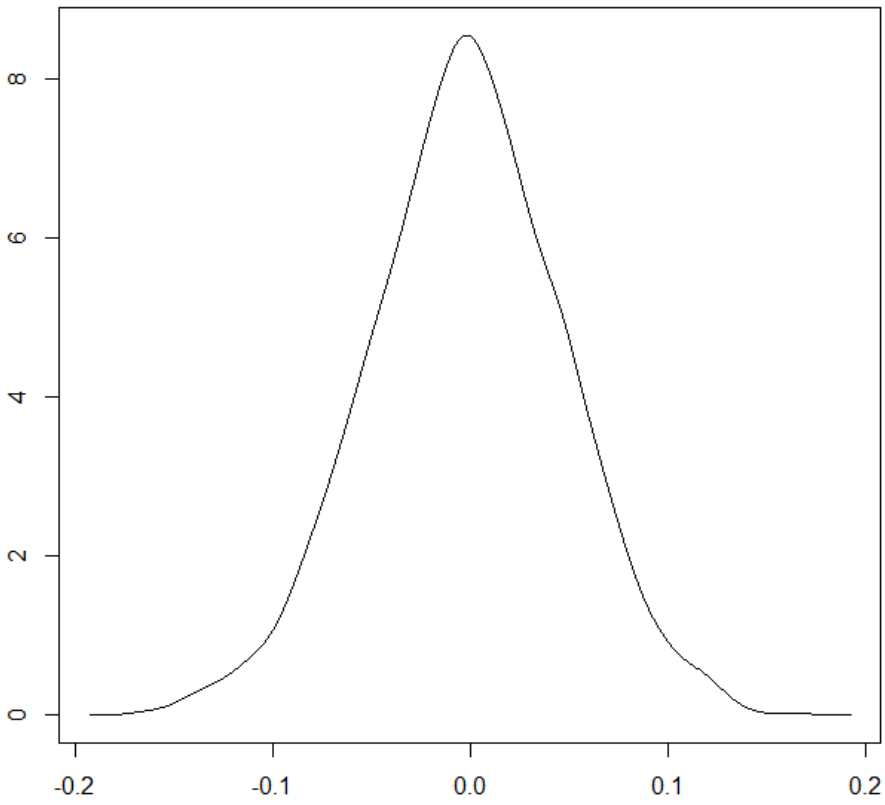


Figure 7: histogram of 1000 Gaussian numbers

```
#define SETUP_SEACH_VARIABLES() \
    auto isMaximizingPlayer = \
        game.activePlayer->getColor() == maximizingPlayer;    \
    float best = (isMaximizingPlayer) ? -INF : INF;           \
    float bestVal;

#define SEARCH_ACTIONS(actions, depthExpr, cmpFunc, toUpdate) \
    for (auto &action : actions) {                            \
        bestVal = recurse(action, depthExpr, alpha, beta);    \
                                                              \
        bestOverallVal = cmpFunc(bestVal, best);              \
        toUpdate = cmpFunc(toUpdate, best);                   \
                                                              \
        if (beta <= alpha) {                                  \
            ++prunedNodes;                                    \
            break;                                            \
        }                                                     \
    }

#define SEARCH_BASED_ON_MAXIMIZING_PLAYER(actions, depthExpr) \
    if (isMaximizingPlayer) {                                 \
        SEARCH_ACTIONS(actions, depthExpr, max, alpha);       \
    } else {                                                  \
        SEARCH_ACTIONS(actions, depthExpr, min, beta);        \
    }
```

Listing 1: alpha beta implementation using c++ preprocessor macros

```cpp
// Use for to pick the right network depending
// on the stage of the game.
for (size_t x = begIdx + 1; x < endIdx; ++x) {

    // Look through the layers in the network
    for (size_t y = 0; y < _layers[x].size(); ++y) {
        size_t prevSize = _layers[x - 1].size();
        float t1 = 0, t2 = 0, t3 = 0, t4 = 0;

        // Loop unrolling for calculating a node
        for (size_t i = 0; i < prevSize; i += 4) {
            // Calculate nodes value
            t1 += _weights[x][y * prevSize + i] *
                _layers[x - 1][i];
            t2 += _weights[x][y * prevSize + i + 1] *
                _layers[x - 1][i + 1];
            t3 += _weights[x][y * prevSize + i + 2] *
                _layers[x - 1][i + 2];
            t4 += _weights[x][y * prevSize + i + 3] *
                _layers[x - 1][i + 3];
        }

        // Total up from unrolled loop
        auto total = t1 + t2 + t3 + t4;

        // Store value for that node
        _layers[x][y] = total / (1 + abs(total));
    }
}

float withPieceCount = _layers[endIdx - 1][0] + pieceCount;

return activation(withPieceCount) * red_factor;
```

Listing 2: Neural network c++ code